Midterm 1 Activity

Stan Drvol and Thomas Kennedy

2/5/2020

Analyzing Recursive Functions' Time Efficiency

In order to gain an insight in the time efficiency of a recursive formula, following these 5 steps is a great place to start.

1. Decide on a parameter (or parameters) indicating an input's size

2. Identify the algorithm's basic operation

3. Check whether the number of times the basic operation is executed can vary on different inputs of same size (if it can, the worst-case, average case, and best-case efficiencies must be investigated separately).

4. Set up recurrence relation (with appropriate initial condition), for the number of times the basic operation is executed.

5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

For our example, we will be using a Binary Search Algorithm

```
// Returns index of x if it is present in arr[l..
    // r], else return -1
    int binarySearch(int arr[], int l, int r, int x)
    {
        if (r >= l) {
            int mid = l + (r - l) / 2;

            // If the element is present at the
            // middle itself
            if (arr[mid] == x)
                return mid;

            // If element is smaller than mid, then
            // it can only be present in left subarray
            if (arr[mid] > x)
                return binarySearch(arr, l, mid - 1, x);

            // Else the element can only be present
            // in right subarray
            return binarySearch(arr, mid + 1, r, x);
        }

        // We reach here when element is not present
        // in array
        return -1;
    }

Source: https://www.geeksforgeeks.org/binary-search/
```

First Step:

Looking at this Algorithm, the amount of times recursed is dependent upon the size of the initial array. For simplicity, we will call its size ‘n’.

Second Step:

The main operation in a search algorithm is the comparison made in the if suites.

Third Step:

Due to the nature of a binary search function, for arrays of the same size, given a different key to find, the worst case would be `O(1)` and the worst and average cases will be approxiately `O(log_2(N))` .

Fourth Step:

To define this recurrence relation, we will assume input array of size n, with the element we are searching for being not present.

The base case `F(0) = 1` , as for an array of length 1 or 0 skips the initial suite and returns -1.

The main function has the relation of `F(n) = F(n/2) + 1`, as the base case is 1, and each time the function is called recursively, it is only searching through half the remaining array.

Fifth Step:

Solving this relation, we first need to transmute our `F(n)` into an easier to deal with `M(n)`. To do this, we have to change it `M(n) = c*M(n-1) + k` for some constants `c, k`. Since `F(n) = F(n/2) +`, intuition tells us that `c = 2^{-1}; k = 1`.

To get from `n to n-2` is demonstrated, and further iterations are left up to the reader as exercise.

```
M(n) = .5M(n-1) + 1
M(n) = .5(.5M(n-1-1) + 1) + 1
M(n) = .5^2M(n-1) + 1 + .5^1
```

Eventually, this will simplify to a sum equal to that of `log_2(n)`, so we can conclude that `O(Binary Search Algorithm) => O(log_2(N))`.